

# Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing

Kirill Garanzha<sup>1</sup> and Charles Loop<sup>2</sup>

<sup>1</sup> Keldysh Institute of Applied Mathematics, Russian Academy of Sciences

<sup>2</sup> Microsoft Research

## Abstract

We present a novel approach to ray tracing execution on commodity graphics hardware using CUDA. We decompose a standard ray tracing algorithm into several data-parallel stages that are mapped efficiently to the massively parallel architecture of modern GPUs. These stages include: ray sorting into coherent packets, creation of frustums for packets, breadth-first frustum traversal through a bounding volume hierarchy for the scene, and localized ray-primitive intersections. We utilize the well known parallel primitives scan and segmented scan in order to process irregular data structures, to remove the need for a stack, and to minimize branch divergence in all stages. Our ray sorting stage is based on applying hash values to individual rays, ray stream compression, sorting and de-compression. Our breadth-first BVH traversal is based on parallel frustum-bounding box intersection tests and parallel scan per each BVH level.

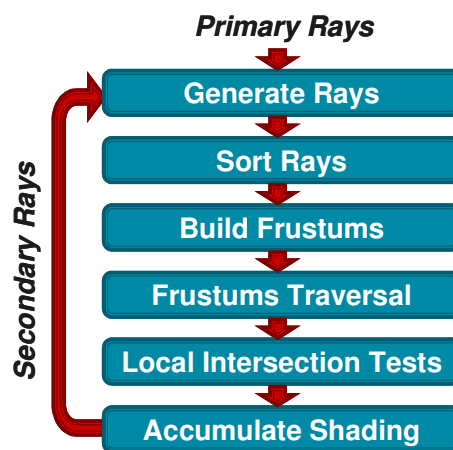
We demonstrate our algorithm with area light sources to get a soft shadow effect and show that our concept is reasonable for GPU implementation. For the same data sets and ray-primitive intersection routines our pipeline is ~3x faster than an optimized standard depth first ray tracing implemented in one kernel.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – Raytracing.

## 1. Introduction

Ray tracing [Whi80; CPC84; Shi00] is a flexible tool to get realistic visual effects such as soft shadows, glossy reflections and global illumination. Ray tracing algorithms rely on a fundamental *trace()* method. The purpose of this *trace()* method is to find the intersection with the closest scene primitive for a given ray. Acceleration structures such as bounding volume hierarchies (BVHs) [KK86] built on scene geometry make the complexity of *trace()* logarithmic with respect to the scene size. Ray tracer execution consists of acceleration structure construction, sampling techniques, acceleration structure traversal, ray-primitive intersection and shading. Ray tracing is a parallel algorithm, since all the rays within the same ray generation can be traced independently.

Our paper is about a faster *trace()* method (that includes acceleration structure traversal and ray-primitive intersection), and not about sampling techniques, shading or acceleration structure construction [Wal07; LGS\*09].



**Figure 1:** Our ray tracing pipeline. Tracing method is decomposed into 4 stages: ray sorting, frustum creation, breadth-first traversal and localized intersection tests.

<sup>1</sup> email: garraill@bk.ru

<sup>2</sup> email: cloop@microsoft.com

Modern desktop GPUs have ~1 TFlops of compute power and ~150 Gb/sec of bandwidth (such as NVIDIA GTX 285). But these compute devices are best suited for streaming data-parallel algorithms with a local and coherent execution and memory access patterns.

In this paper we propose a novel packet-based ray tracing pipeline where the *trace()* method is represented in four stages: ray sorting into coherent packets, creation of the frustums for packets, breadth-first frustum traversal through a bounding volume hierarchy for the scene, and localized ray-primitive intersections (see Fig. 1). Our first contribution is the ray sorting stage that is based on applying hash values to individual rays, ray stream compression, sorting and decompression. Our second contribution is the stack-less breadth-first BVH traversal that is based on parallel frustum-bounding box intersection tests and parallel scan per each BVH level.

Our work was inspired by Arvo and Kirk's 5D ray tracing [AK87]. In that work a volume in 5D space (3D for origin and 2D for direction) is used to represent a collection of rays. The initial 5D volume is decomposed into a tree of hypercubes that are linked to lists of scene primitives. Using this tree, the rays (5D points) are associated with a hypercube and tested for intersection with a local list of primitives. In contrast, we extract the coherent packets of required rays; efficiently build the list of primitives per each packet using a 3D bounding volume hierarchy for the scene (and all the tracing stages are executed on a GPU).

## 2. Background

### 2.1 GPU Computing Model

Modern GPUs are composed of several independent SIMD cores [NVIDIA; NBGS08]. Each core can execute multiple threads in parallel that communicate via shared on-chip memory. The GPU supports thousands of parallel threads that execute 1 simple program (kernel) for different data elements. The invocation of these kernels is organized in the host program (running on the CPU). GPU threads are subdivided into blocks (each block is executed on the single GPU core). Shared memory is much faster than global GPU memory and can be used for inter-thread communication within a single block of threads. Each block of threads is organized into several **warps** (bundles of 32 threads) which execute a single kernel instruction for the entire warp of threads.

**SIMD/SIMT.** SIMT (single instruction multiple threads) is a superset of SIMD where thread divergence is handled by hardware. This feature simplifies programming but in order to get high GPU compute utilization one should organize the execution so that all the threads within a warp make the same branching decisions and access coherent memory locations. If the code of the kernel contains conditions and if some threads within a warp take different branches then they will execute both code paths bringing

all the other threads of the warp with them. This case provides multi-branching and complicated memory access pattern, resulting in a loss of compute efficiency.

**Parallel Scan and Segmented Scan.** We utilize the well known parallel primitives scan and segmented scan [SHG08] in order to process irregular data structures. Parallel scan (or prefix sum) for a given array results in the output array where the  $i^{th}$  element is the sum of all the previous elements of the input array (including the  $i^{th}$  element for inclusive scan and excluding the  $i^{th}$  element for exclusive scan). Parallel segmented scan is the same as scan, but this procedure computes the sum of all the previous elements within a segment of input array. The input array may contain arbitrary sized segments that are concatenated in a contiguous array. The segments of the input array are specified by an additional array of head flags where all the elements set to one denote the bases of segments; all the zero elements denote the tails of segments. [CUDPP] provides a library of these parallel primitives.

### 2.2 Ray Tracing Review

**Coherent Ray Tracing.** Efficient ray-packet generation and tracing is a challenging task. Recent research has focused significant effort on improving high coherence situations [GPSS07; WBS07; HM08; BW09]. These techniques are successful for primary rays, hard shadows or soft shadows with small area lights. This comes from grouping coherent rays together, bounding them within a tight frustum and then performing the same traversal process for the group of rays. Such a technique amortizes the average computational and memory pattern access costs for a single ray. Ray packet grouping is based on the original screen-space layout of rays.

**Breadth-first ray tracing and ray reordering.** Breadth first ray tracing was first investigated by Hanrahan [Han86]. The idea behind breadth-first ray tracing is to form a set of rays and intersect each BVH node against this set during top down traversal. The set of rays is gradually reordered for deeper BVH levels [GR08; ORM08] (if there is no ray-box intersection then the ray reference is eliminated from the set of rays that descends to the node's children). This algorithm amortizes the cost of node access pattern among the rays. A stack is used to defer intersection tests for neighboring nodes within a BVH. Our breadth-first BVH frustum traversal is based on the full parallel scan for all frustums (and rays) per each BVH level and does not use a stack. Boulos et al. [BWB08] described a combination of CPU-based ray reordering techniques for improved coherence and effect on performance.

**GPU Ray Tracing.** Most GPU ray tracers are implemented by mapping each ray to one thread. Each ray needs a separate hierarchy traversal stack with a depth equal to the maximum depth of the scene hierarchy. The stack is usually stored in a private thread memory (local memory in

CUDA) that is slower than shared memory since it is mapped to global GPU memory [NVIDIA]. Several works [HSHH07; GPSS07] described ways to eliminate or mitigate stack usage in a GPU ray tracer. But these approaches have no solution for the warp-wise multi-branching problem that was analyzed by Aila and Laine [AL09]. This problem was mitigated by using persistent threads that fetch the ray tracing task per each idle warp of threads. Some warps within a block of threads become idle if one warp executes longer than others. In our ray tracing pipeline we eliminate this warp-wise multi-branching at expense of a long pipeline and special ray sorting. Roger et al. [RAH07] presented a GPU ray-space hierarchy construction process based on screen-space indexing. Rays were not actually sorted for better coherence.

### 3. GPU Ray Tracing Pipeline

In order to map ray tracing to efficient GPU execution we decompose ray tracing into 4 stages: ray sorting, frustum creation, breadth-first traversal, and localized ray-primitive intersections (see Fig. 1).

Ray sorting is used to store spatially coherent rays in consecutive memory locations. Compared to unsorted rays, the tracing routine for sorted rays has less divergence on a wide SIMD machine such as GPU. Extracting packets of coherent rays enables tight frustum creation for packets of rays. We explicitly maintain ray coherence in our pipeline by using this procedure.

We create tight frustums in order to traverse the BVH using only frustums instead of individual rays. For each frustum we build the spatially sorted list of BVH-leaves that are intersected by the frustum. Given that the set of frustums is much smaller than the set of rays, we perform breadth-first frustum traversal utilizing a narrower parallel scan per each BVH level.

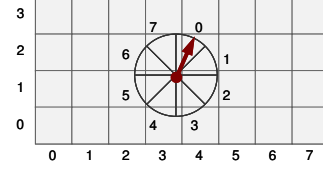
In the localized ray-primitive intersection stage, each ray that belongs to the frustum is tested against all the primitives contained in a list of sorted BVH-leaves captured in a previous stage.

#### 3.1 Ray Sorting

Our ray sorting procedure is used to accelerate ray tracing by extracting coherence and reducing execution branches within a SIMD processor. However, the cost of such ray sorting should be offset by an increase in performance. We propose a technique that is based on compression of key-index pairs. Then we sort the compressed sequence and decompress the sorted data.

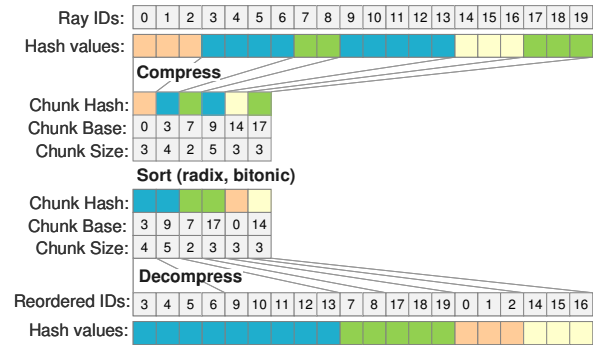
**Ray hash.** We create the sequence of key-index pairs by using the ray id as index, and a hash value computed for this ray as the key. We quantize the ray origins assuming a virtual uniform 3D-grid within scene's bounding box. We also quantize normalized ray directions assuming a virtual

uniform grid (see Fig. 2). We manually specify the cell sizes for both virtual grids (see section 5.1). With quantized components of the origin and direction we compute cell ids within these grids and merge them into a 32-bit hash value for each ray. Rays that map to the same hash value are considered to be coherent in the 3D-space.



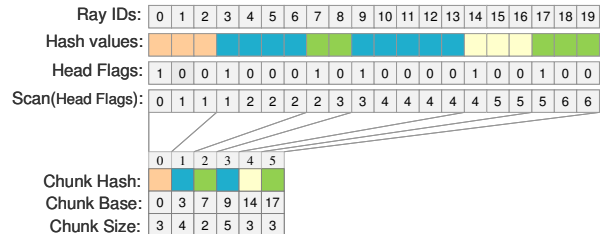
**Figure 2:** The quantization of ray origin and direction is used to compute a hash value for a given ray.

**Sorting.** We introduce a “compression – sorting – decompression” (CSD) scheme (see Fig. 3) and explicitly maintain coherence through all the ray bounce levels. Coherent rays hit similar geometry locations. And these hit points form ray origins for next-generation rays (bounced rays). There is a non-zero probability that some sequentially generated rays will receive the same hash value. This observation is exploited and sorting becomes faster. The compressed ray data is sorted using radix sort [SHG09].



**Figure 3:** The overall ray sorting scheme.

**Compression.** We create the array *Head Flags* equal in size to the array *Hash values*. All the elements of *Head Flags* are set to 0 except for the elements whose corresponding hash value is not equal to the previous one (see Fig. 4). We apply an exclusive scan procedure [SHG08] to the *Head Flags* array.



**Figure 4:** Compression example.

We then perform data compaction into *Chunk Base* and *Chunk Hash* arrays: for each  $Head\ Flags_i = 1$  we write the value  $i$  into position of *Chunk Base* array specified by  $Scan(Head\ Flags)_i$ . Analogously, we build *Chunk Hash* array. The values of *Chunk Size* elements are equal to differences between neighboring *Chunk Base* elements.

**Decompression.** When the compressed data is sorted we apply an exclusive scan procedure to the *Chunk Size* array (see Fig. 5). We initialize the array *Skeleton* with ones, and the array *Head Flags* with zeroes (the sizes of both arrays are equal to *Hash values* array). Into positions of the array *Skeleton* specified by  $Scan(Chunk\ Size)$  we write the corresponding values of *Chunk Base* array. Into positions of the array *Head Flags* specified by  $Scan(Chunk\ Size)$  we write ones. We then apply an inclusive segmented scan [SHG08] to array *Skeleton* considering the *Head Flags* array that specifies the bounds of data segments. The result of the segmented scan is the array of reordered (sorted) ray ids corresponding to their hash values.

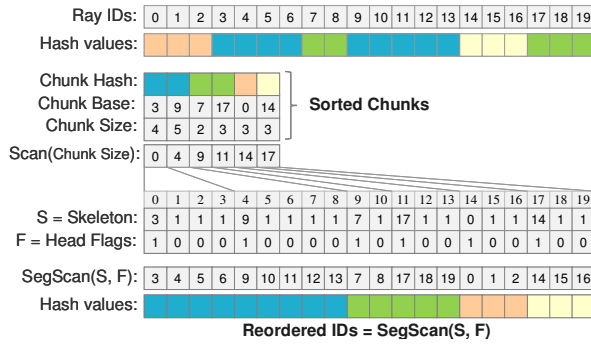


Figure 5: Decompression example.

**Decomposition: packet ranges extraction.** We would like to create packets of coherent rays no larger than some capacity (e.g.,  $MaxSize = 256$ ). First, we extract the base index and range of each cell that contains the chunk of rays with the same hash value. In order to do this we apply the compression procedure described above to the array of sorted rays. As a result each element of *Chunk Size* represents the number of rays assigned to the corresponding grid cell (see Fig. 6).

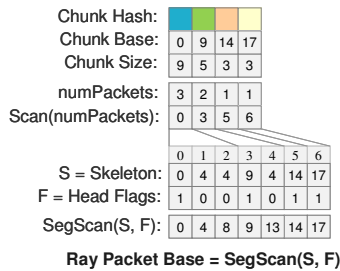


Figure 6: Decomposition example. On this example each chunk is decomposed into the packets of  $MaxSize = 4$ .

We create the array *numPackets* where  $numPackets_i = (ChunkSize_i + MaxSize - 1) / MaxSize$  and then scan this array. All the values of *Skeleton* are initially set to  $MaxSize$  and all values of *Head Flags* are set to zero. Into positions of the array *Skeleton* specified by  $Scan(numPackets)$  we write the corresponding values of array *Chunk Base*. Into positions of the array *Head Flags* specified by  $Scan(numPackets)$  we write ones. As in the decompression procedure, we apply an inclusive segmented scan to array *Skeleton* considering the *Head Flags*. The result of this segmented scan is the array of base indices for each ray packet, the size of a ray packet is found as the difference of consecutive bases.

### 3.2 Frustum Creation

Once the rays are sorted and packet ranges extracted, we build a frustum for each packet. As in the work [ORM08], we define the frustum by using a dominant axis and two axis-aligned rectangles. The dominant axis corresponds to the ray direction component with a maximum absolute value. For the coherent rays of a packet this axis is assumed to be the same. The two axis-aligned rectangles are perpendicular to this dominant axis and bound all the rays of the packet (see Fig. 7).

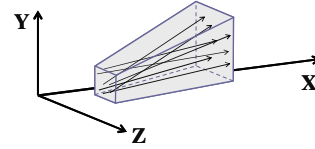


Figure 7: Frustum is defined by dominant axis  $X$  and two axis-aligned rectangles.

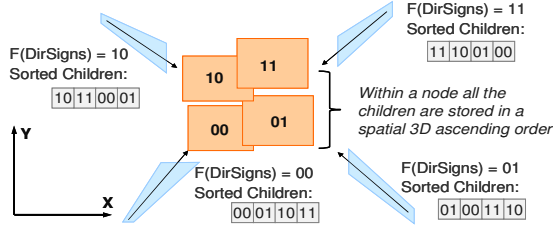
We implemented the frustum creation in a single CUDA kernel where each frustum is computed by a warp of (32) threads. Shared memory is used to compute the valid interval along the dominant axis and base rectangles for all the rays in a packet.

### 3.3 Breadth-First Frustum Traversal

We perform breadth-first frustum traversal through the BVH with the arity equal to eight. The binary BVH is constructed on the CPU and  $2/3^{rd}$ s of tree levels are eliminated and an Octo-BVH is created (all the nodes are stored in a breadth-first storage layout). Each BVH-node is represented with 32 bytes: six float values for the axis-aligned bounding box (AABB), one 32-bit integer value represents the block of children (3 bytes for the base offset of the block and 1 byte for the number of children), and one 32-bit integer for the spatial order of children within this node. All the children within the node are sorted in a spatial 3D ascending order (see Fig. 8).

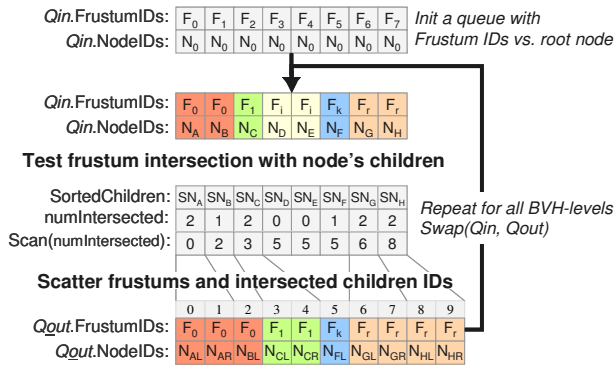
**Per frustum child ordering.** For each frustum, a 3-bit value of  $F(DirSigns)$  is computed that corresponds to the sign bits of the average frustum's ray direction. The spatial order of node's children along the frustum direction is

computed using an xor-operation:  $SortedChildren[i] = i \wedge F(DirSigns)$ . See Fig. 8 for examples. Though this ordering may not be exact for all the cases, it is simple and sufficient for our purposes.



**Figure 8:** Four examples of node's children ordering along the frustum are presented in a 2D projection.

**Traversal.** For simplicity, we allocate a large memory block for the working queues (e.g. 4-8 million elements) used in the traversal procedure. We initialize the array  $Qin.FrustumIDs$  with frustum ids, and the array of  $Qin.NodeIDs$  with the BVH root node id (i.e. with a zero). At each iteration step (for each level) of the traversal procedure we perform intersection tests for each frustum corresponding to  $Qin.FrustumIDs_i$  with all the children of the BVH-node specified by  $Qin.NodeIDs_i$  (using AABB-Frustum culling). The number of intersected children is saved in  $numIntersected_i$ . Spatially sorted 3-bit local offsets of the intersected children are packed in a single word  $SortedChildren_i$  (see Fig. 9). We apply an exclusive scan procedure to the  $numIntersected$  array. Into the chunk of the array  $Qout.FrustumIDs$  specified by offset equal to  $Scan(numIntersected)_i$  and length equal to  $numIntersected_i$  we propagate the value of  $Qin.FrustumIDs_i$ . Into the corresponding chunk of  $Qout.NodeIDs$  we write global ids of the children that were packed in  $SortedChildren_i$ . A global id for  $k$ -th child of node is computed as the sum of  $k$  and the base offset of node's children. Then we swap the pointers  $Qin$  and  $Qout$ . This iterative procedure is repeated for all the levels of the BVH. The leaf-nodes of unbalanced tree are considered to be the children of themselves and we bring them to the bottom level of the tree. The number of levels is reduced by using Octo-BVH.



**Figure 9:** Breadth-first frustums' traversal example.

After all traversal iterations are finished the  $Qout.FrustumIDs$  array contains the irregular chunks with the same values (frustum ids were propagated per chunk). The corresponding chunks of  $Qout.NodeIDs$  array contain the references to the intersected BVH leaves that are approximately spatially sorted along the frustums. We extract the array of distinct frustum ids (active frustums) and the corresponding ranges of leaf-chunks with a compression procedure applied for  $Qout$  (see section 3.1, replace *Hash values* with  $Qout.FrustumIDs$  in example Fig. 4).

### 3.4 Localized Intersection Tests

When we obtain the array of active frustum ids and corresponding ranges of leaf-chunks per frustum we decompose all the active frustums into chunks of 32 rays max (**ray warps**) – since the numbers of rays per frustum may not be equal. This decomposition is done analogously to the example in Fig. 6. Each ray warp is mapped to a CUDA thread warp execution (32 threads in a warp). All rays within a warp share the same frustum id that determines the computations and memory reads (see Fig. 10). This kind of execution eliminates multi-branching within a CUDA warp of threads and only ray masking remains. In this stage we exploit the sorted list of leaves per each frustum. For all rays in each ray warp the intersections are analyzed for the closest triangles first and ray distance parameters are updated. Using updated ray distance parameters we avoid intersection tests for occluded triangles (using ray-AABB test). We use CUDA persistent threads [AL09] in order to balance the workload since the frustums may capture irregular chunks of intersected leaves during traversal stage.

```
while(ray warps are available) { // persistent
    RayWarp = fetch_next_warp(); // threads [AL09]
    Ray = fetch_ray(RayWarpBase + threadIdx.x);
    FrustumId = frustum_id(RayWarp);
    for(all leaves(FrustumId))
        if(Ray intersects AABB(Leafi)) // mask rays
            for(all primitives(Leafi)) // coherent reads
                intersect Ray with a primitive;
}
```

**Figure 10:** CUDA kernel of localized ray-primitive intersection tests.  $threadIdx.x$  belongs to interval  $[0..32)$ .

## 4. Benchmark Implementation

We test the new ray tracing pipeline for primary rays (at 1024x768 resolution) and soft shadow rays (with area light source and 16 samples maximum per shade point).

In the ray sorting stage we sort only ray origins for soft shadow samples (see Fig. 2). The virtual uniform grid for ray sorting has a user-specified size of the cell (cells are cubical). This size is selected as the fraction of scene's bounding box extents:  $CellSize = UserCellFraction * SceneBboxDiagonal$ . Ray directions are sorted per each origin according to the stratified sampling over the light source domain. For big area light sources the rays that hit 4 or 16 different stratum are assigned to 4 or 16 different frustums (however some rays may share the same origins).

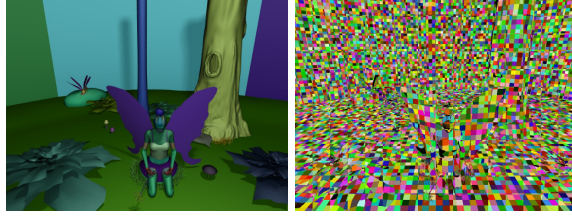


Within a compression-sorting-decompression (CSD) scheme we use a full 32-bit radix sort [SHG09]. It is possible to apply a faster partial sorting procedure (e.g. bitonic sort from CUDA SDK) that locally sorts data within the equal chunks of compressed ray buffer. This procedure may extract reasonable, but not perfect ray coherence.

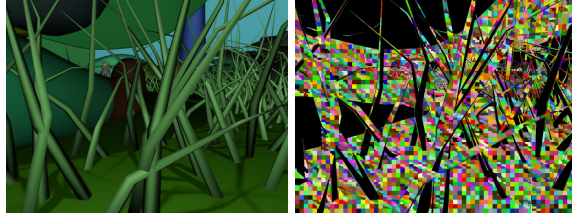
Primary rays are indexed and sorted according to a screen-space Z-curve (256 rays per primary frustum).

We use the Utah Fairy Forest (174K triangles), Conference (280K triangles), and Sponza (68K triangles) as the test scenes for our algorithm (all tested viewpoints are presented on Fig. 11). We build a binary BVH on the CPU for these models using a binning algorithm [Wal07]; we stop recursive construction and create a leaf when a BVH-node contains less than (or equal to) 10 triangles. As a ray-triangle intersection test we use the scheme of Möller-Trumbore [MT97].

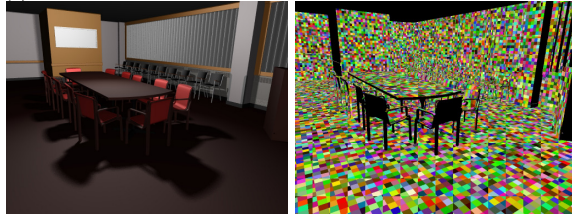
(a) Fairy Forest (simple view)



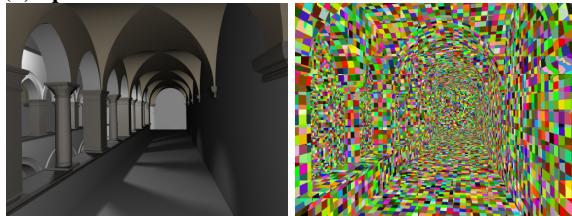
(b) Fairy Forest (complex view)



(c) Conference



(d) Sponza



**Figure 11:** Benchmark scenes and tested viewpoints. Right column: an example of soft shadow frustum origins.

All measurements were done using an NVIDIA GTX 285 and all the kernels were compiled using CUDA 2.2 [NVIDIA]. We compare the new ray tracing pipeline with our implementation of “*persistent speculative while-while*” (the most efficient) ray tracing kernel described by Aila and Laine [AL09]. For both methods the same data-sets, BVH, intersection tests and viewpoints are used (for the new pipeline we convert the binary BVH to the Octo-BVH).

When we measure ray tracing performance (milliseconds, rays/second) for the new pipeline we take into account only tracing-specific stages: “*sort rays*”, “*build frustums*”, “*traversal*”, “*intersection tests*” (see Fig. 1).

In our [AL09] implementation we replace three of our logic stages “*build frustums*”, “*traversal*”, “*intersection tests*” with a single “*trace*” kernel described in [AL09]. In all performance measurements we take into account only “*trace*” timings, but still this method works with sorted rays (for these rays CUDA thread warps should execute with less divergence). Ray sorting timings are not taken into account when we evaluate [AL09] performance.

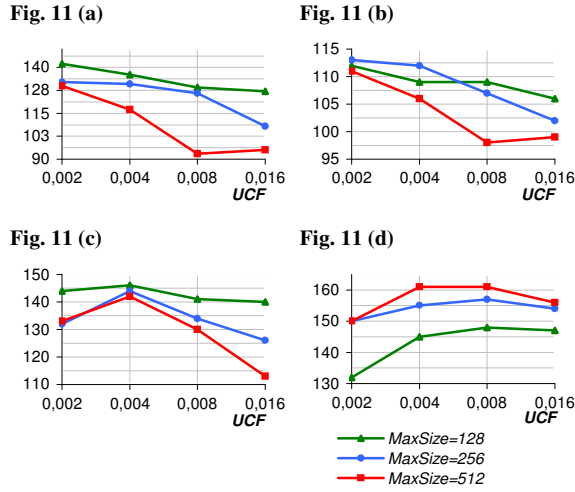
## 5. Results and Analysis

### 5.1 Ray Quantization Parameters Evaluation

UCF	Fig.11(a)	Fig.11(b)	Fig.11(c)	Fig.11(d)
<b>MaxSize=128</b>				
0.002	681k / 108k / 31	432k / 67k / 31.9	859k / 119k / 30	721k / 77k / 30.8
0.004	668k / 97k / 31.6	444k / 66k / 31.9	691k / 91k / 30.9	610k / 66k / 31.4
0.008	690k / 91k / 31.9	461k / 65k / 31.9	618k / 78k / 31.7	557k / 60k / 31.8
0.016	709k / 89k / 31.9	467k / 65k / 31.9	640k / 74k / 31.9	546k / 59k / 31.9
<b>MaxSize=256</b>				
0.002	521k / 80k / 30.5	232k / 35k / 31.9	945k / 116k / 29	512k / 50k / 30.9
0.004	444k / 57k / 31.5	233k / 34k / 31.9	572k / 62k / 30.8	378k / 37k / 31.5
0.008	445k / 48k / 31.9	254k / 33k / 31.9	438k / 44k / 31.6	318k / 32k / 31.8
0.016	500k / 46k / 31.9	258k / 33k / 31.9	447k / 39k / 31.9	303k / 30k / 31.9
<b>MaxSize=512</b>				
0.002	430k / 66k / 30.6	130k / 19k / 31.8	792k / 96k / 29.5	494k / 45k / 30.8
0.004	317k / 35k / 31.5	134k / 18k / 31.9	431k / 44k / 31.0	283k / 25k / 31.5
0.008	313k / 26k / 31.8	144k / 17k / 31.9	303k / 26k / 31.6	189k / 14k / 31.8
0.016	328k / 24k / 31.9	148k / 16k / 31.9	294k / 21k / 31.9	171k / 16k / 31.9

**Table 1:** Ray quantization parameters and working statistics for soft shadows: the number of leaves references captured for all frustums by traversal stage / the number of frustums / average number of rays per ray warp (frustums are decomposed into ray warps in intersection stage). This data depends on the size of frustums; the size of each frustum depends on the UserCellFraction and MaxSize. A bigger value of UCF (UserCellFraction) denotes a bigger cubic cell in a virtual grid (according to which ray origins are quantized). MaxSize denotes the maximum capacity (in rays) per each frustum. This data is presented for the fixed area light source. Performance is given on Fig. 12.

Traversal and intersection statistic for different ray quantization parameters are given in Table 1. E.g.  $LS$  = “the number of leaves captured for all frustums” and  $FS$  = “the number of frustums” represent the working queue size of the breadth-first traversal. A relation  $LS/FS$  is the average number of leaves captured per each frustum. For all quantization parameters (given in Table 1) a value of this relation is around 10. Given the fact that we build the BVH with ~10 triangles per leaf the maximum number of triangle intersection tests per ray in our benchmarks should be ~100. But all the leaves are sorted along the frustum direction and we perform ray masking (i.e. if the AABB of the leaf is not intersected, see Fig. 10) so the actual number of ray-triangle intersection tests can be much lower.

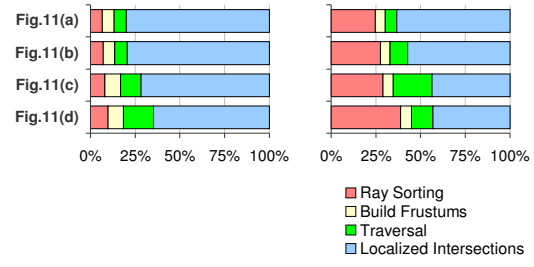


**Figure 12:** Ray quantization parameters and performance in million rays / second for soft shadow rays (bigger numbers are better). Parameters meaning and stats are given in Table 1.

Actual performance of ray tracing is given in Fig. 12 (for the viewpoints presented in Fig. 11) and it is not clear what parameters are the best for all scenes. However selecting  $MaxSize=256$  and  $UserCellFraction=0.004$  seems to be robust for high performance ray tracing and leads to relatively small working queues. We use these parameters for all the following measurements and comparisons.

## 5.2 Ray Tracing Pipeline Stages

Fig. 13 presents the time spent in different stages of our pipeline for soft shadows with a fixed light source. For the left chart 1024x768 elements are sorted in a ray sorting stage. This stage takes ~6ms for all scenes and includes hash value computation, compression, 32-bit radix sort, decompression, frustum ranges extraction. For the right chart 1024x768x16 elements are sorted in ~40ms with a CSD scheme (including all the supplementary routines). In contrast, only the 32-bit radix sort (without CSD) for 1024x768x16 elements takes ~80ms.

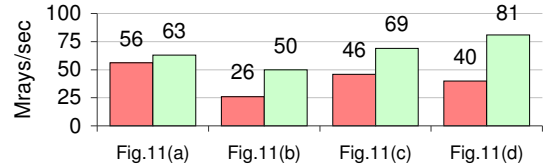


**Figure 13:** Time spent in logic stages of ray tracing pipeline for soft shadow rays. Left chart: 16 shadow rays were generated per primary hit point. Right chart: 1 shadow ray was generated per primary hit point (with 4x4 per pixel antialiasing). For the right chart data there are 16 shadow samples per pixel (and we sort 16x more ray origins overall than for the left chart data).

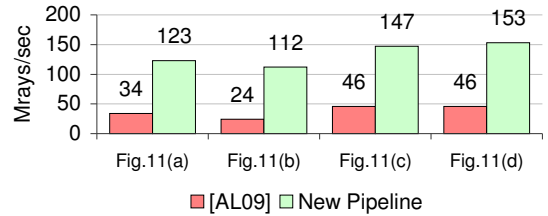
## 5.3 Comparison with a Depth-first Ray Tracing

The charts in Fig. 14 represent our pipeline in comparison to our implementation of the Aila and Laine approach [AL09]. The gap between two approaches is bigger for soft shadow rays that are less coherent since we reduce warp-wise branches in our ray tracing pipeline (we have only ray masking in intersection stage, see Fig. 10).

### Primary rays (at 1024x768):

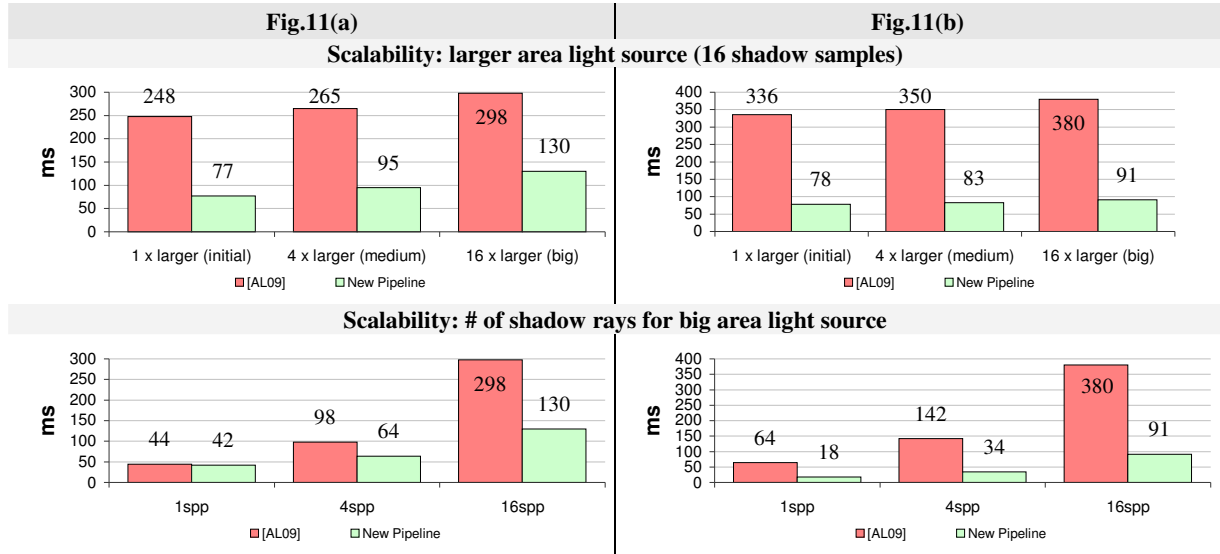


### Soft Shadow rays (at 1024x768x16 samples):



**Figure 14:** Performance comparison of our ray tracing pipeline and our implementation of [AL09] (bigger numbers are better). See Fig. 11 for viewpoints.

Performance measurements (rays per second) of the depth-first ray tracing implementation [AL09] may be different from results published in this paper. We build the BVH with another algorithm without tessellating large triangles; we use different triangle intersection tests, different viewpoints and sampling techniques. But the input data and all these intersection routines are the same for our comparisons. Careful splitting of large triangles may provide significant speedup for ray tracing (e.g. 2-3x [DK08;



**Figure 15:** Timings comparison of our pipeline and our implementation of [AL09] in a Fairy Forest (see Fig. 11a,b) (smaller numbers are better). These charts represent the difference between two approaches in shadow tracing for larger area light sources (with the same # of rays) and for the different # of samples (with a fixed light source).

EG07]). Our scenes contain large triangles. Low level optimizations are not applied yet for our new pipeline and the implementation of [AL09].

#### 5.4 Scalability

The charts on Fig. 15 represent ray tracing timings for larger light source (less coherent ray directions) and different sampling rates (less dense packets for our pipeline). The relative speedup of our approach compared to [AL09] is increased for the viewpoint on Fig. 11b. This viewpoint was analyzed in our benchmarks since the depth of geometry is highly varying in a screen-space as well as for shadow rays. This setup can be considered as the hard case for packet-based ray tracers (or ray tracers that rely on wide SIMD/SIMT units only). A difficult part for these techniques is that individual rays may make different decisions during traversal and follow different branches. But all these individual decisions will bring any other rays within a packet or warp of threads to the useless paths for them. This case provides multi-branching and complicated memory access pattern that affects ray tracing performance. In our approach we explicitly maintain a linear execution without multi-branching but within a longer pipeline.

#### 5.5 Discussion

The new ray tracing pipeline provides the possibility to trace relatively big packets of rays and perform efficient view-independent queries using a breadth-first frustum traversal. Memory access patterns for breadth-first traversal are coherent as we perform operations in parallel for each BVH level (and the BVH is stored in a breadth-first layout). The warp-wise execution and memory access pattern

of the intersection stage are also coherent according to the explicit work-load organization.

Since we store all the leaf references for all the frustums the memory consumption may be considerable (and we also store the rays). But this consumption may be reduced through using a screen-space tiling (send reasonably big tiles to render on the GPU) or even frustum depth tiling.

A bad case for our algorithm (and for many others) would be if one frustum captures all the leaves of the scene's BVH and other frustums capture nothing during breadth-first traversal. This would cause a very unbalanced workload for the intersection stage that will be not hidden or amortized by persistent threads. In this case it is possible to replicate the ray hit information (hit distance parameter and primitive id) of this frustum into  $n$  instances. Then each of these instances perform parallel intersection tests with  $1/n^{\text{th}}$  of the leaves for this frustum. After all the intersections are tested then all results are merged by segmented parallel reduction with a min-operation. This reduction can be implemented much like segmented scan [SHG08].

**BVH traversal.** For our pipeline we convert a binary BVH to the Octo-BVH since this operation reduces the height of the tree by a factor of 2-3 and reduces the number of calls for the scan operation. Octo-BVH also increases the number of AABB-frustum culling tests and texture fetches per each traversal thread (8 node children are tested per thread). Overall, the breadth-first traversal stage with the Octo-BVH is 2x faster than with a binary BVH.

We actually implemented the depth-first traversal stage for created frustums (where each frustum selects  $K$  leaves



and the intersections are performed in another stage). But depth-first traversal was 5x slower than a breadth-first one.

**Comparison to CPU ray tracer.** We also implemented a CPU ray tracer with our packet assembling approach (sorting only origins of shadow samples). These origins were sorted using a binning approach for grids assigned to screen-space tiles. We employed CPU-friendly algorithms: tile-based multi-core parallelism, depth-first traversal, SIMD instructions, advanced triangle culling techniques and other state of the art CPU optimizations that are very similar to the approaches described in [BW09]. Although no details are presented on the charts, our new ray tracing pipeline running on a GTX 285 is ~4x faster than a CPU-friendly implementation running on a Core 2 Quad 2.4GHz Q6600. We used almost the same scene setup to compare both versions (the same scenes and BVHs, but different viewpoints and execution routines).

## 6. Conclusion and Future Work

In this paper we have presented a novel ray tracing pipeline using CUDA. This pipeline consists of multiple data-parallel stages where the warp-wise multi-branching is eliminated. We rely on the ray sorting procedure and explicitly maintain coherent execution within all the stages of our pipeline. We have proposed a simple compression-sorting-decompression (CSD) technique in order to accelerate the ray sorting stage. We have proposed a fast stackless breadth-first frustum traversal algorithm that supports view-independent queries by using a full parallel scan of each BVH level.

An advantage of our work is that it is a software pipeline that runs on existing GPU (this means flexibility). We have also shown that the ray tracing executed in our pipeline is faster than current state of the art GPU or CPU ray tracers. Though our results and implementation are still preliminary, we expect performance improvement for Whitted ray tracing and path tracing. A complete solution should take a large number of rays (e.g. a set of arbitrary rays from a single ray-generation level) at input and provide intersection information at output (for shading stage).

We are going to extend the application of our CSD techniques. First, we will implement a faster GPU-based acceleration structure (BVH) builder using the Z-curve indexing, sorting, compression, BVH-construction [LGS\*09] and decompression. Second, we are going to accelerate the stream reordering for deferred shading [HLJH09].

Finally, we are going to integrate a Reyes-style adaptive smooth surface subdivision [EML09] into our ray tracing pipeline that will rely on frustum-patch subdivision criterion. Assuming the view-independent ray tracing queries, this feature should increase the visual quality of geometry appearance.

## Acknowledgments

We thank the University of Utah for the Fairy scene, Anat Grynberg and Greg Ward for Conference scene and Marko Dabrovic for Sponza Atrium scene. We are grateful to the anonymous reviewers for their valuable comments.

## References

- [AK87] ARVO J., KIRK D.: Fast ray tracing by ray classification. *ACM SIGGRAPH Computer Graphics* 21, 4 (1987), 55–64.
- [AL09] AILA T., LAINE S.: Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the High Performance Graphics* (2009).
- [BW09] BENTHIN C., WALD I.: Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. In *Proceedings of the High Performance Graphics* (2009).
- [BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive Ray Packet Reordering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).
- [CPC84] COOK R., PORTER T., CARPENTER L.: Distributed Ray Tracing. *Computer Graphics (Proceedings of SIGGRAPH 84)* 18, 3 (1984), 137–144.
- [CUDPP] CUDPP: CUDA data parallel primitives library, (2009). <http://www.gpgpu.org/developer/cudpp/>.
- [DK08] DAMMERTZ H., KELLER A.: The Edge Volume Heuristic – Robust Triangle Subdivision for Improved BVH Performance. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).
- [EG07] ERNST M., GREINER G.: Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007).
- [EML09] EISENACHER C., MEYER Q., LOOP C.: Real-Time View-Dependent Rendering of Parametric Surfaces. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (2009).
- [GR08] GRIBBLE C.-P., RAMANI K.: Coherent Ray Tracing via Stream Filtering. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).
- [GPSS07] GÜNTHER J., POPOV S., SEIDEL H.-P. and SLUSALLEK P.: Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007).
- [HAN86] HANRAHAN P.: Using caching and breadth first traversal to speed up ray tracing. In *Proceedings of the Graphics Interface* (1986).
- [HM08] HUNT W., MARK W.-R.: Ray-Specialized Acceleration Structures for Ray Tracing. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).

- [HLJH09] HOBEROCK J., LU V., JIA Y., HART J.C.: Stream Compaction for Deferred Shading. In *Proceedings of the High Performance Graphics* (2009).
- [HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU ray tracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007).
- [KK86] KAY T., KAJIYA J.: Ray tracing complex scenes. In *Proceedings of SIGGRAPH* (1986), 269–278.
- [LGS\*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. In *Proceedings of Eurographics* (2009).
- [MT97] MÖLLER T., TRUMBORE B.: Fast, minimum storage ray triangle intersection. *JGT*, 2(1): 21-28, (1997).
- [NBGS08] NICKOLLS J., BUCK I., GARLAND M., SKADRON K.: Scalable parallel programming with CUDA. *Queue* 6, 2 (2008), 40–53.
- [NVIDIA] NVIDIA Corporation. NVIDIA CUDA Programming guide (2009).
- [ORM08] OVERBECK, R., RAMAMOORTHY, R., MARK, W.-R.: Large Ray Packets for Real-time Whitted Ray Tracing. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).
- [RAH07] ROGER D., ASSARSSON U., HOLZSCHUCH N.: Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU. In *Proceedings of Eurographics Symposium on Rendering* (2007).
- [SHG08] SENGUPTA S., HARRIS M., GARLAND M.: Efficient parallel scan algorithms for GPUs. *NVIDIA Technical Report NVR-2008-003* (December 2008).
- [SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore GPUs. In *Proceedings 23rd IEEE International Parallel and Distributed Processing Symposium* (2009).
- [Shi00] SHIRLEY P.: Realistic Ray Tracing. *AK Peters, Ltd.*, 2000.
- [WAL07] WALD I.: On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing* (2007).
- [WBB08] WALD I., BENTHIN C., BOULOS S.: Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-branching BVHs. In *Proceedings of the 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008).
- [WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1 (2007), 1–18.
- [WHI80] WHITTET T.: An improved Illumination Model for Shaded Display. *Communications of the ACM* 23, 6 (1980), 343–349.